Theoretical Computer Science 57 (1988) 327-330
North-Holland

NOTE

AN INHERENTLY ITERATIVE COMPUTATION OF ACKERMANN'S FUNCTION

Jerrold W. GROSSMAN

Department of Mathematical Sciences, Oakland University, Rochester, MI 48309, U.S.A.

R. Suzanne ZEITMAN

Department of Computer Science and Engineering, Oakland University, Rochester, MI 48309, U.S.A.

Communicated by A. Salomaa
Received December 1986

Abstract. The Ackermann function is defined recursively by $A(0, n) = n+1$; $A(i, 0) = A(i-1, 1)$ for $i > 0$; and $A(i, n) = A(i-1, A(i, n-1))$ for $i, n > 0$. An iterative algorithm for computing $A(i, n)$ is presented. It has $O(i)$ space complexity and $O(iA(i, n))$ time complexity, both of which are much smaller than the corresponding quantities for an algorithm based directly on the recursive definition.

Of theoretical interest in computer science is a function $A: \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ attributed to Wilhelm Ackermann. To logicians, it is the canonical example of a recursive function that is not primitive recursive. A sort of "inverse" of the function plays a role in measuring the time complexity of algorithms for set manipulation [7], finding short cycles of even lengths in graphs [4], and job scheduling with two processors [2], to name but a few applications. Because its definition is highly recursive in form, it has been used to measure performance of implementations of recursive subroutine calls in programming languages [8].

In this note we present a new way to look at the Ackermann function. Our approach is iterative rather than recursive, and we will view its computation as an exercise in counting (in the naive sense), rather than an exercise in recursion. Under the appropriate model our algorithm for computing values of the Ackermann function uses space proportional to the indexing variable and time almost proportional to the value of the function, considerably better than the corresponding performance of the usual algorithms.

We begin with the nested recursive definition.

Definition. The *Ackermann function* $A: \mathbf{N} \times \mathbf{N} \to \mathbf{N}$, where $\mathbf{N}$ is the set of nonnegative integers, is defined by

$$A(i, n) = \begin{cases} n+1 & \text{if } i = 0, \\ A(i-1, 1) & \text{if } i > 0 \text{ and } n = 0, \\ A(i-1, A(i, n-1)) & \text{if } i > 0 \text{ and } n > 0. \end{cases}$$

We call i the indexing variable.

It is an easy exercise to obtain the following explicit formulas, which indicate how explosively the values $A(i, n)$ grow:

$$A(0, n) = n+1, \qquad A(1, n) = n+2,$$
$$A(2, n) = 2n+3, \qquad A(3, n) = 2^{n+3}-3, \tag{*}$$
$$A(4, n) = 2^{2^{\cdot^{\cdot^{2}}}} - 3 \quad (n+3 \text{ twos}).$$

We feel compelled to digress for a historical comment that may eliminate some of the confusion surrounding the Ackermann function. Some authors (Tarjan [7], for example) use a variant of the function A, in which the formulas (*) come out somewhat simpler at the expense of a more complicated definition. Ackermann's original function (which he called φ) actually had three arguments, the last of which was the indexing variable. It satisfied the formulas

$$\varphi(a, b, 0) = a + b, \qquad \varphi(a, b, 1) = ab,$$
$$\varphi(a, b, 2) = a^b, \qquad \varphi(a, b, 3) = a^{a^{\cdot^{\cdot^{a}}}},$$

with b a's in the exponent, etc. That hardly anyone seems to have consulted Ackermann's original paper [1] is clear not only from the fact that the definition of φ is given wrong, but also from the fact that the title of the paper is misspelled by nearly every author who references it. As far as we can determine, the standard definition of A we give here was first used by Raphael Robinson [6] twenty years after Ackermann's paper, although a nearly identical function was defined by Rózsa Péter [5] seven years after Ackermann. In fact, David Hilbert [3] described Ackermann's function two years before his student Ackermann published the proof that it was recursive but not primitive recursive (Hilbert credited the proof to Ackermann). Maybe it should bear Hilbert's name as well.

The function A is usually viewed in the context of the formulas (*). To compute $A(i, n)$ in general one either copies the definition into a recursive procedure, or else uses a stack in the obvious way to mimic the recursion (see Algorithm 1).

It is easy to show that the maximum length of the stack in Algorithm 1 is $A(i, n)$, as long as $i > 0$. Thus, if we assume a model of computation in which any natural number requires one word of storage, then the space required to compute $A(i, n)$ is $O(A(i, n))$. Furthermore, since small values of A must get recalculated repeatedly, the time requirement is much worse.

```
Procedure Stackermann(i, n)
  {stack is initially empty}
  push i onto stack
  push n onto stack
  while stack has more than one item do begin
    pop n-current from stack
    pop i-current from stack
    if i-current = 0 then push n-current + 1 onto stack
    else if n-current = 0 then begin
      push i-current - 1 onto stack
      push 1 onto stack
      end
    else begin
      push i-current - 1 onto stack
      push i-current onto stack
      push n-current - 1 onto stack
      end
    end while
  pop n-current from stack
  return n-current {the value of A(i, n)}
  end Stackermann
```

Algorithm 1. Using a stack to compute $A(i, n)$ recursively.

Reflection on the definition of A shows that the sequences $A(i, \cdot)$ for $i = 1, 2, \ldots$ are simply more and more rarefied increasing sequences of integers, each a subsequence of the previous one, where the particular elements chosen are determined by the subsequence being generated. Figure 1 indicates what happens for the first few values of i.

To calculate values of the Ackermann function, it will be sufficient to keep track of where we are in each subsequence (array *next* in Algorithm 2) and where we need to reach before transferring the value just calculated to the next subsequence (array *goal* in Algorithm 2). Filling in the details, we get Algorithm 2, the proof of whose correctness is left to the reader.

If we again assume that any integer can be stored in one word of memory, then the space requirement is negligible; the following theorem is clear.

Theorem 1. *Algorithm 2 requires* $O(i)$ *space to compute* $A(i, n)$.

Furthermore, since essentially all the algorithm does is to count by ones (variable *value* in Algorithm 2), from 1 to $A(i, n)$, with a little extra bookkeeping, we have the following result on its time requirement; again the proof is omitted.

$i = 0$: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, ...
$i = 1$: 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, ...
$i = 2$: 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, ...
$i = 3$: 5, 13, 29, 61, 125, 253, 509, 1021, 2045, 4093, 8189, 16381, 32765, 65533, ...
$i = 4$: 13, 65533, ...

Fig. 1. The subsequences $A(i, \cdot)$ for $0 \leq i \leq 4$.

```
Procedure Ackermann(i, n)
  {next and goal are arrays indexed from 0 to i, initialized so that next[0] through next[i]
   are 0, goal[0] through goal[i-1] are 1, and goal[i] is -1}
  repeat
    value ← next[0]+1
    transferring ← true
    i-current ← 0
    while transferring do begin
      if next[i-current] = goal[i-current] then goal[i-current] ← value
                                           else transferring ← false
      next[i-current] ← next[i-current]+1
      i-current ← i-current+1
      end while
  until next[i] = n+1
  return value {the value of A(i, n)}
  end Ackermann
```

Algorithm 2. Computing $A(i, n)$ iteratively.

Theorem 2. *Algorithm 2 requires* $O(iA(i, n))$ *time to compute* $A(i, n)$.

As a practical indication of the relative efficiency of these two algorithms (along with the obvious recursive procedure), we timed the computation of $A(3, 5)$ on an Apple *IIe* computer running UCSD Pascal programs of all three procedures. The recursive version overflowed its system stack and could not obtain the answer. Algorithm 1 required over three minutes of computation time (and took over 42,000 passes through the while loop) to obtain the answer of 253. Algorithm 2 took three seconds. In fact, Algorithm 2 computed $A(4, 1) = 65533$ in less than 12 minutes, whereas Algorithm 1 would have run for several months.

References

[1] W. Ackermann, Zum Hilbertschen Aufbau der reellen Zahlen, *Math. Ann.* **99** (1928) 118–133.

[2] H. N. Gabow, An almost linear algorithm for two-processor scheduling, *J. Assoc. Comp. Mach.* **29** (1982) 766–780.

[3] D. Hilbert, Über das Unendliche, *Math. Ann.* **95** (1926) 161–190.

[4] B. Monien, The complexity of determining a shortest cycle of even length, in: *Proc. 8th Conf. on Graph-theoretic Concepts in Computer Science* (*WG 82*), Neunkirchen (1982) 195–208.

[5] R. Péter, Konstruktion nichtrekursiver Funktionen, *Math. Ann.* **111** (1935) 42–60.

[6] R. M. Robinson, Recursion and double recursion, *Bull. Amer. Math. Soc.* **54** (1948) 987–993.

[7] R.E. Tarjan, Efficiency of a good but not linear set union algorithm, *J. Assoc. Comp. Mach.* **22** (1975) 215–225.

[8] B.A. Wichmann, Ackermann's function: a study in the efficiency of calling procedures, *Nordisk Tidskr. Informationsbehandling* (*BIT*) **16** (1976) 103–110.